Analyzing Energy Friendly Steady State Phases of Dynamic Application Execution in Terms of Sparse Data Structures

E.G. Daylight*
IMEC vzw
Kapeldreef 75
3001 Leuven
Belgium
voudheus@imec.be

S. Wuytack
C. Ykman-Couvreur
IMEC vzw
Kapeldreef 75
3001 Leuven
Belgium
wuytack@imec.be
couvreur@imec.be

F. Catthoor†
IMEC vzw
Kapeldreef 75
3001 Leuven
Belgium
catthoor@imec.be

ABSTRACT

In the past decades, data structure analysis was mainly done at a high level of abstraction in the computer science community. For instance, choosing a linked list as a data structure as opposed to an array for a specific situation, was mainly motivated from a performance point of view under the implicit assumption that the computer platform (that had to run the software) consisted out of one monolithical, physical memory. In the context of mobile, embedded devices, energy consumption is as important as performance. In addition to this, the assumption of one monolithical memory is outdated for many (if not all) current-day platforms! Clearly, there is a need to improve the choices that are made during data structure analysis given specific knowledge of the memory hierarchy of the platform under investigation.

We show how memory related energy consumption can heavily be reduced by taking into account the access behaviour of the application on the one hand and the available on-chip and off-chip memory space on the other hand. We do this by exploiting the sparseness that is present in one *steady state* of the data structure under investigation. Analytical results show that energy reductions of a factor of 8.7 are feasible in comparison to common data structure implementations. We trade these gains off with on-chip memory space consumption of a custom memory architecture.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: [composite structures, linked representations]; C.4 [**Computer Systems Organization**]: Performance of Systems—*performance attributes*

Keywords

Energy consumption, on-chip memory footprint, partitioned data structure

1. INTRODUCTION

There is a need to improve the choices that are made at the level of data structure analysis given specific knowledge of the memory hierarchy of the platform under investigation. This is because high-level, design decisions -such as data structure analysis- that are made early in the design trajectory (of software/hardware systems) have a large impact on the eventual quality of the implementation. We will demonstrate this issue by generating low-energy consuming, partitioned data structures in the context of an embedded, multimedia game application. We would however like to stress that we are addressing a fundamental issue here, namely that of data structure analysis. The preliminary methodology we present is applicable in many (if not all) applications due to the fact that we address trivial access operations such as lookup, iterate, insert, and remove behaviour in the context of a small, on-chip scratch-pad memory and a large, off-chip memory of a custom memory organization.

In ongoing work we are applying the same philosophy towards low-cost software that has to be executed on a predefined memory architecture. In this context, the on-chip memory corresponds to the cache of the system and the off-chip memory to an SDRAM. We use our non-trivial data structure implementations to impose on current-day used (hardware controlled) caching schemes.

In this paper, we only present work related to custom memory organizations. Also, we do not explain in detail our analytical model and we do not go into detail about the game application we have used as our driver. The focus of this paper is to show the practical value of our approach. We combine simple but powerful knowledge from the application side on the one hand and the platform side on the other hand and very easily construct energy efficient implementations at the end of the day.

All results we show are based on the assumption that the data structure we analyze has a sparseness of $20 : 128$. This means that

*Also at Computer Science Department, Katholieke Univ. Leuven.
†Also professor at Katholieke Univ. Leuven.

ISLPED'02 August 12-14, 2002, Monterey, California, USA.

an average number of $C = 20$ valid data elements is stored in the data structure which can contain a total of 128 data elements. This assumption is based on experiments that we have done in IMEC on a 3D computer game.

This one specific value of C corresponds to one specific steady state phase of execution when viewed in terms of sparseness. During the whole execution, different steady states are present and corresponding transitions from one steady state data structure implementation to another have to be made at run time. We do not cover these issues here.

Our results are also based on the following. Each data element is uniquely specified by seven key bits ($log_2 128 = 7$). The size of a data element is assumed to be $R = 224$ bits (and the size of a pointer is $P = 8$ bits). Different data elements may be stored in the data structure during different moments of the one and only steady state we analyze.

Related work is described in Section 2. Terminology and our energy model are introduced in Section 3. The methodology and design tool are explained in Section 4. We present our analytical results and conclusions in Section 5.

2. RELATED WORK

The work presented in this article is inspired by [15] and [17]. There are however two main differences.

First of all, we analyze the *software* implementation of an energy efficient data structure as opposed to explicitly designing and using specific, configured, physical memories (*hardware)* of the embedded system. Redesigning the custom memory organization is not feasible in our context. We assume that the embedded platform is already designed and fully functional on the Internet. Stated otherwise, software that migrates to the embedded system will transform its critical data structures in conformance with the physical memory architecture of the target platform.

The second difference with the referenced work is the kind of application that is being investigated. As opposed to optimizing heavy data-oriented networking applications (e.g. routers [16]), this article shows that relatively small data-storing applications, in the Internet context, can perform better as well by applying similar data structure optimizations. We intentionally analyze small data structures to show that our approach is also relevant for those. This is because *well-written*, modular software encapsulates small pieces of functionality and corresonding data structures from other pieces of code.

Low-power issues are the main focus in the following two research communities: dynamic power management (see [1] for more references) and dynamic voltage scheduling [7]. However, data management related issues are not or hardly covered in these communities.

Traditional data structure analysis can be found in [3, 6, 10]. The work presented here is strongly related to that of virtual memory management [4, 5, 13]. The difference is that this work explores the possibilities on how to represent a data structure assuming that all the physical memory is already allocated. The optimizations based on choosing a good dynamic (de)allocation strategy are complementary to the optimizations that are described here.

Other related work is [9, 8, 12]. The emphasis of the corresponding authors lies on exploiting the physical characteristics of the architecture (e.g. cache line size) based on profiling information to reduce the execution time of the (memory intensive) application. We acknowledge these as being very powerful optimizations but they are complementary to the design choices we present here. In addition to this, we focus on energy consumption of the application as opposed to the more common notion of execution speed.

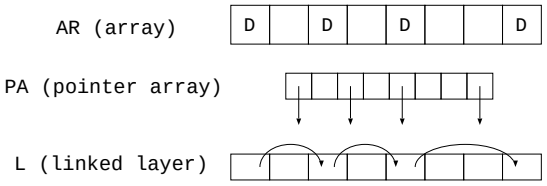

Figure 1: Three primitive data structures that can be combined to produce more complex data structures. 'D' denotes a valid data element. An arrow denotes a valid pointer value.

3. TERMINOLOGY AND ENERGY MODEL

We make a distinction between *data structure* on the one hand and *partitioned data structure* on the other hand. The term data structure corresponds to that used in the literature while a partitioned data structure is a data structure that is mapped onto the on-chip/off-chip memory hierarchy in a specific way. For instance, the LPAcomp[1] data structure, shown in Figure 3, is mapped in two different ways in this article. This leads to two different partitioned data structures: the LPAcompH (Figure 4) and the LPAcompC (Figure 5).

In our analytical approach, we assume the presence of an embedded system consisting of one on-chip memory and one off-chip memory. We have compared the energy consumption for one on-chip access (0.48nJ) and one off-chip access (4.29nJ) for two memories of 0.18μ technology. We have used a version of the Cacti model [14] that produces energy estimates to compute the energy consumption values. The on-chip SRAM memory has a 32-bit bus and 16 KB memory size. The off-chip SRAM memory has a 32-bit bus and a 200 KB memory size. The ratio, in energy consumption, between an on-chip access and an off-chip access is 8.94.

4. METHODOLOGY AND DESIGN TOOL

We have developed a design tool that accepts the following as input: data storage information, access behaviour, and memory architecture cost functions.

The data storage information includes the average number (C) of stored data elements in the data structure under investigation, the maximum number of key bits (K) that are needed to be able to uniquely characterize every data element, the size of a pointer (P) in bits, and the size of a data element (R) in bits. Specifying the value K is equivalent to stating how many data elements can be stored in the data structure. Indeed, the capacity of the data structure is equal to 2^K. The tool examines only one steady state phase or thus only one value of C. As stated previously, we analyze the steady state of $C = 20$. The tool is configured so that one memory access is needed to read or write the contents of a data element (denoted by 'D' in the figures). For instance, for the array shown in Figure 1, only one access is needed to read or write a data element (D).

The access behaviour describes how the data structure is most often accessed. The tool contains a predefined set of behaviours (e.g. look up, iterate, insert, remove). Look up behaviour corresponds to the retrieval of a data element, given a specific key value. Iteration behaviour corresponds to the traversal of all the data elements that are stored in the data structure regardless of the associated key value of every data element. Insert behaviour corresponds to the insertion of a data element. Remove behaviour corresponds to the removal of a data element.

[1]The term 'LPAcomp' is an abbreviation for 'linked pointer array complex data structure'.

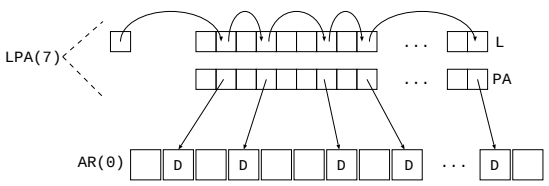

Figure 2: The LPA(7)AR(0). This is a three-layered data structure which is not key-splitted. $2^7 = 128$ different data elements (D) can potentially be stored in the data structure.

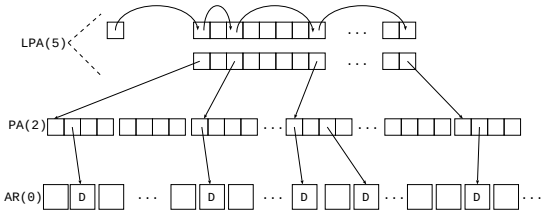

Figure 3: The LPA(5)PA(2)AR(0) data structure (alias LPAcomp). This is a partially key-splitted data structure.

For the memory architecture cost functions, we multiply the number of on-chip accesses with $0.48nJ$ and the number of off-chip accesses with $4.29nJ$. Adding the two obtained values gives the energy consumption.

The output of the tool is a specification of the optimal partitioned data structure (e.g. a specification of the partitioned data structure in Figure 5).

The methodology is composed of two steps: (1) combining primitive data structures and applying key splitting; and (2) partitioning.

4.1 Combining Primitive Data Structures

Primitive data structures are combined to produce a multilayered data structure.

Three different primitive data structures are used by the tool (cfr. Figure 1). They are the array (AR), the pointer array (PA), and the linked array (L).

An example of combining primitive data structures to form a multilayered data structure is combining the linked layer (L(7)), the pointer array (PA(7)), and the AR(0). This results in the LPA(7)AR(0) which is shown in Figure 2.

The linked layer allows an easy traversal through the data structure. Looking up an element in the LPA(7) is simply done by indexing into the pointer array layer and retrieving the data element which is being pointed to.

4.2 Key Splitting

Key splitting is applied on zero, one, or all layers of the multi-layered data structure. This results in an increased multi-layering of the data structure such as the one shown in Figure 3.

Key splitting [17] has three different contributions.

First, key splitting exploits the *sparseness* that is present in the data structure [17].

Second, depending on the specific behaviour (e.g. iteration), key splitting can contribute to decreasing the number of accesses. Consider for instance the AR(7) on the one hand and the LPAcomp (cfr. Figure 3) on the other hand. The AR(7) needs 128 accesses in the case of iteration behaviour even though only 20 data elements are actually stored. The LPAcomp needs only 72 accesses for iteration behaviour. (We omit the calculations for brevity.)

Third, key splitting can decrease the wasted space of the available on-chip memory. In other words, key splitting allows the available on-chip memory space to be used more economically. This corresponds to the second step of the methodology (see below). Consider again the LPAcomp and the AR(7). The LPAcomp can be horizontally partitioned into the LPAcompH (cfr. Figure 4).

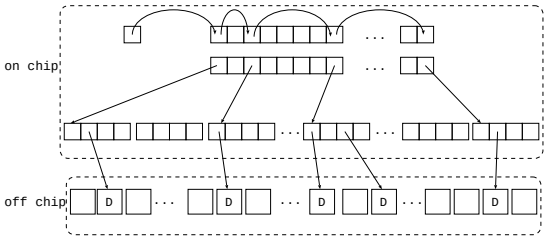

Figure 4: Horizontal partitioning of the LPAcomp: the LPAcompH.

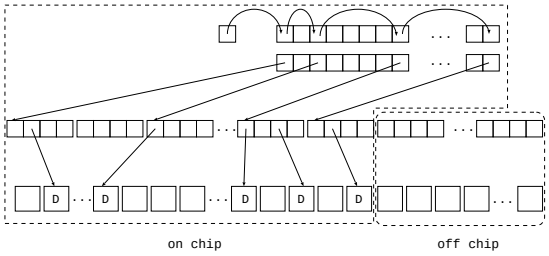

Figure 5: A possible complex partitioning of the LPAcomp: the LPAcompC. This is an abbreviation for [5296][[24928]LPAcomp which means that a total of 5296 bits of on-chip memory space is consumed and 24928 bits of off-chip memory space.

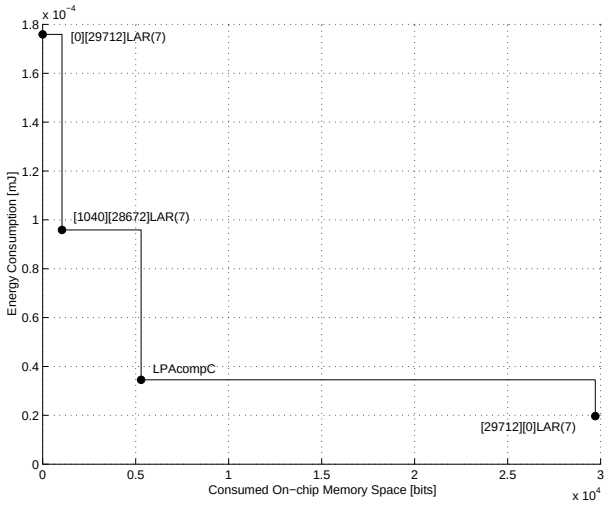

Figure 6: Pareto curve in the context of iteration behaviour (C=20, K=7). The energy consumption for one iteration is shown versus the amount of consumed on-chip memory space. Each point on the Pareto curve represents a partitioned data structure. The constraints of the Pareto curve are (i) the available on-chip memory space, (ii) the amount of energy that is consumed on the partitioned data structure when looking up, inserting, or removing a data element, (iii) a constant time for look up, insert, and removal of a data element. Only the first constraint is shown explicitly (on the x-axis) on the graph.

This results in 52 on-chip accesses and 20 off-chip accesses during iteration. The AR(7) on the other hand is (usually) too large in relationship to the available on-chip memory space. This results in placing the AR(7) fully off-chip, implying that 128 off-chip accesses are needed for one iteration. In this example, the AR(7) consumes 28672 bits of memory space, which has to either be placed all on chip or all off chip. The LPAcompH however, consumes only 1552 bits of on-chip memory space.

4.3 Partitioning

In the second step, the multilayered data structure is *partitioned* into the on-chip memory and the off-chip memory (assuming that only these two memories are present in the embedded system).

Consider once again the data structure shown in Figure 3. A possible horizontal partitioning of this data structure is shown in Figure 4. In this example, the first three layers of the data structure are placed in the on-chip memory. The fourth layer is placed in the off-chip memory.

Vertical partitioning of the LPAcomp (or any data structure in general) is possible too but we omit the discussion here for brevity.

Figure 5 represents a complex partitioning of the LPAcomp. The average number (20) of data elements are placed on-chip. All accesses are on-chip accesses. The drawback is the increase in the on-chip memory space consumption when compared to the LPAcompH.

5. RESULTS AND CONCLUSIONS

Analytical results are shown in Figure 6. Note that only a selected set of optimal partitioned data structures are shown in the figure (e.g. we show only one complex partitioned data structure). The *Pareto optimal curve* allows a software designer to trade-off energy consumption with consumed on-chip memory space.

We have shown that by exploiting (a) the sparseness of the data structure, (b) the access behaviour of the data structure, and (c) the on-chip memory space configuration, large reductions in energy consumption (of a factor of 8.7) are feasible. Non-trivial, partitioned data structures are recommended as opposed to ordinary data structure implementations. The optimization techniques presented in this article are directly applicable to small data structures and easily extendible to large data structures and corresponding larger physical memories (in which even larger gains may be expected).

6. REFERENCES

[1] L.Benini, G.DeMicheli, *"Dynamic Power Management Design Techniques and CAD Tools"*, 1998, Kluwer Academic Publishers, ISBN 0-7923-8086-X.

[2] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecapelle, *"Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design"*, Kluwer Academic Publishers, 1998.

[3] T.H.Cormen, C.E.Leiserson, R.L.Rivest, *"Algorithms"*, 1998, Prentice-Hall of India, ISBN-81-203-1353-4.

[4] JL.da SilvaJr, C.Ykman-Couvreur, M.Miranda, K.Croes, S.Wuytack, G.de Jong, F.Catthoor, D.Verkest, P.Six, H.De Man, "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", *Proc. 35th ACM/IEEE Design Automation Conf.*, San Fransisco CA, pp.76-81, June 1998.

[5] D.Haggander, P.Liden, L.Lundberg, "A Method for Automatic Optimization of Dynamic Memory Management in C++", *Proc. ICPP 01. the 30th International Conference on Parallel Processing*, Valencia, Spain, pp.489-498, Sep. 2001.

[6] C.A.R.Hoare, "Proof of Correctness of Data Representations", *Acta Informatica* 1, 271-281, 1972 Springer Verlag.

[7] N.K.Jha, "Low power system scheduling and synthesis", *IEEE Int. Conf. on Computer-Aided Design*, Nov. 2001.

[8] T.Kistler, M.Franz, "Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance", *ACM Transactions on Programming Languages and Systems*, Vol.22, No.3, pp.490-505, May. 2000.

[9] T.Kistler, M.Franz, "Continuous Program Optimization: Design and Evaluation", *IEEE Transactions on Computers*, Vol.50, No.6, pp.549-565, Jun. 2001.

[10] D.E.Knuth, *"The Art of Computer Programming"*, Vol. 3, Addison-Wesley, 1973.

[11] P.Marchal, C.Wong, et al., "Dynamic memory oriented transformations in the MPEG4 IM1-player on a low power platform", *Proc. Intnl, Wsh. on Power Aware Computing Systems (PACS)*, Cambridge MA, pp.31-40, Nov. 2000.

[12] S.A.McKee, W.A.Wulf, J.H.Aylor, R.H.Klenke, M.H.Salinas, S.I.Hong, D.A.B.Weikle, "Dynamic Access Ordering for Streamed Computations", *IEEE Transactions on Computers*, Vol.49, No.11, pp.1255-1270, Nov. 2000.

[13] P.R.Wilson, M.S.Johnstone, M.Neely, D.Boles, "Dynamic Storage Allocation: A Survey and Critical Review", *Proc. 1995 Int'l. Workshop on Memory Management*, Kinross, Scotland, UK,September 27-29, 1995, Springer Verlag LNCS.

[14] S.J.E.Wilton, N.P.Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model", *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 5, May, 1996.

[15] S.Wuytack,F.Catthoor,H.DeMan, "Transforming Set Data Types to Power Optimal Data Structures", *Proc. IEEE Intnl. Workshop on Low Power Design*, Laguna Beach CA, pp.51-56, April 1995.

[16] S.Wuytack, J.L.daSilvaJr, F.Catthoor, G.deJong, C.Ykman-Couvreur, "Memory Management for Embedded Network Applications", *IEEE Transactions on Computer-Aided Design* , Vol. 18, No.5, May, 1999.

[17] C.Ykman-Couvreur, J.Lambrecht, D.Verkest, F.Catthoor, H.De Man, "Exploration and Synthesis of Dynamic Data Sets in Telecom Network Applications", *Proc, 12th ACM/IEEE Intnl. Symp. on System-Level Synthesis (ISSS)*, San Jose CA, pp.125-130, Dec. 1999.